An Evolutionary Approach to Software Modularity Analysis

Sunny Huynh and Yuanfang Cai
Department of Computer Science
Drexel University
Philadelphia, PA
{sh84, yfcai}@cs.drexel.edu

Abstract

Modularity determines software quality in terms of evolveability, changeability, maintainability, etc. and a module *could be a vertical slicing through source code directory structure or class boundary. Given a modularized design, we need to determine whether its implementation realizes the designed modularity. Manually comparing source code modular structure with abstracted design modular structure is tedious and error-prone.*

In this paper, we present an automated approach to check the conformance of source code modularity to the designed modularity. Our approach uses design structure matrices *(DSMs) as a uniform representation; it uses existing tools to automatically derive DSMs from the source code and design, and uses a genetic algorithm to automatically cluster DSMs and check the conformance.*

We applied our approach to a small canonical software system as a proof of concept experiment. The results supported our hypothesis that it is possible to check the conformance between source code structure and design structure automatically, and this approach has the potential to be scaled for use in large software systems.

1. Introduction

Modularity in design is immensely important for software systems, determining software quality in terms of evolveability, changeability, maintainability, etc. [2, 15]. However, even if the design of a software system is well modularized it can be difficult to determine if the source code faithfully implements the designed modularity. In this paper, we present an approach to addressing this problem by automatically checking the conformance of source code modular structure to design modular structure.

To perform the conformance analysis, we first need a uniform representation capable of capturing the modular structure of both source code and high level design. We observe that *design structure matrices* are effective in fulfilling this need. In DSM, the rows and columns are labeled with variables, representing both environment factors and design dimensions. Cells in a DSM are marked according to dependencies among design decisions and environmental influence. Columns and rows within a DSM can be arranged so that the variables are clustered into abstract modules, forming blocks of cells along the diagonal of the DSM.

According to Baldwin and Clark [3] and Sullivan et al. [17], the block-diagonal structure of a DSM captures how a software design is decomposed into modules, revealing the level of coupling and cohesion through the density of dependence marks within a block and between blocks. Moreover, DSM modeling allows the designer to cluster variables in different ways, each representing one view of decomposition. For example, Sullivan et al. [17] showed that by clustering a DSM into an environment cluster, a design rule cluster, and hidden modules, Parnas's information hiding criteria can be precisely captured. Lattix [1], for example, recovers DSM models from Java sourse code, and clusters them by Java class packages.

Cai's design level DSM derivation tool, Simon [5], allows designers to formally model software designs and derive DSMs from logical models. Lattix [1], a source code level DSM derivation tool, allows designers to automatically recover a DSM from an existing software implementation. With these tools, both the high level design structure and source code structure can be represented by DSMs, allowing for a uniform assessment of their modular structures. Ideally, if the implementation exactly conforms to the design, each clustered block of variables in the source code DSM can be mapped to a block of variables in the design level DSM.

However, design DSMs and source code DSMs work at different levels of abstraction. A design DSM usually needs higher level of abstraction to obtain the full picture of the system, while a source code DSM usually uses classes or other program constructs as variables labeling the rows and columns. For a large software system, there could be hun-

First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)
0-7695-2967-4/07 $20.00

dreds or thousands of variables in the source code DSM but much fewer design variables in the high level design DSM. Take Liebeherr's HyperCast, a peer-to-peer networking system with thousands of lines of code and hundreds of files, for example. The HyperCast design DSM derived by Simon [4] consists of about 30 variables, while the source code DSM derived by Lattix consists of over 250 variables. Clustering the 250 variable DSM in different ways and checking the conformance to the design DSM is tedious and error-prone.

To address this problem, we first formalize the conformance checking problem mathematically, and then use a genetic algorithm [7] to automate the clustering of variables in a DSM and assess the conformance between design modularity and source code modularity. A genetic algorithms is a search technique in the field of evolutionary computing, which finds approximate solutions to optimization problems by simulating the evolution of a population of potential solutions. Given two DSMs, one at the design level and the other at the source code level, our genetic algorithm takes one DSM as the optimal goal and searches for a best clustering method in the other DSM that maximizes the level of isomorphism between the two DSMs.

The rest of this paper is arranged as follows. Section 2 lays out the mathematical representations and foundations to the conformance checking problem. Section 3 elaborates on the technique of the genetic algorithm. In section 4, as a proof of concept demonstration, we run our tool on a small software system. Lastly, section 6 concludes with our results from the proof of concept.

2. Problem Representation

This section presents the mathematical formulation of the conformance checking problem that will be processed by the genetic algorithm.

First, we define $\mathbb{X}$ be the set of all components in the implemented software system; that is, the set of all variables from the source code DSM. $D_X \subseteq \mathbb{X} \times \mathbb{X}$ is the set of dependencies among the software components, such that $\forall i, j \in \mathbb{X}\ (i,j) \in D_X$ if and only if component i depends on component j. That is $(i,j) \in D_X$ if and only if the cell representing the dependency of variable i on variable j is marked in the source code DSM.

We define $\mathbb{Y}$ to be the set of all design level components, meaning the variables in the design level DSM. $D_Y \subseteq \mathbb{Y} \times \mathbb{Y}$ is the set of dependencies among the design level components, such that $\forall i, j \in \mathbb{Y}\ (i,j) \in D_Y$ if and only if component i depends on component j. That is $(i,j) \in D_Y$ if and only if the cell representing the dependency of variable i on variable j is marked in the design level DSM.

DSMs can be considered as attributed, directed graphs in which the variables serve as attributes for the vertices and the dependency structure portrayed by the DSM serve as directed edges. That is, each vertex of the graph represents a variable of the DSM and there exists a directed edge in the graph if and only if there is a marked cell in the DSM between those two variables. Accordingly, we transform both the source code DSM and the design level DSM into graphs for purpose of mathematical processing.

In the next subsections, we first defined one of the two DSMs as a *sample graph*, then define the other as a *model graph*, and finally define their *conformance criteria*.

2.1. Sample Graph

A *sample graph* has more vertices and needs to be clustered, usually transformed from a source code DSM. Mathematically, a sample graph is defined as an attributed, directed graph $\vec{S} = (V_S, E_S)$ (i.e. V_S is the set of vertices and E_S is the set of edges). Attributes are assigned to each vertex to associate it with a particular variable of the source code DSM via the bijection $k_S : V_S \rightarrow \mathbb{X}$. There is an edge in the graph for each dependency between variables of the source code DSM, and there are no other edges (i.e. $(v_i, v_j) \in E_S \Leftrightarrow (v_i, v_j) \in D_X$).

2.2. Model Graph

We usually use *model graphs*, also attributed, directed graphs, to represent high level design DSMs with more abstract, but fewer in number, design variables: $\vec{M} = (V_M, E_M)$ (i.e. V_M is the set of vertices and E_M is the set of edges). Attributes are assigned to each vertex to associate it with a particular variable of the design level DSM via the bijection $k_M : V_M \rightarrow \mathbb{Y}$. There is an edge in the graph for each dependency between variables of the design level DSM, and there are no other edges (i.e. $(v_i, v_j) \in E_M \Leftrightarrow (v_i, v_j) \in D_Y$).

2.3. Conformance Criteria

To determine the conformance of the source code modularity to the high level design modularity, we cluster the variables of the sample graph and therefore form a new graph $\vec{C} = (V_C, E_C)$ (i.e. V_C is the set of vertices and E_C is the set of edges), called the *conformance graph*. Each vertex of the conformance graph is associated with a cluster of variables from the sample graph. This is rigorously defined via the bijection $k_C : V_C \rightarrow V_S/\sim$, such that the clustering of variables forms a quotient set of V_S with respect to an arbitrary equivalence relation $\sim$. Edges are defined by the dependencies among components within each

equivalence class such that $\forall v_i, v_j \in V_S/\sim \ (v_i, v_j) \in E_C \Leftrightarrow \exists (u_i \in v_i, u_j \in v_j)\ (u_i, u_j) \in E_X$.

The more conforming the source code modularity is to the design modularity, the closer to isomorphic the conformance graph and model graph will be. We define the mapping of vertices from the conformance graph to the model graph as the bijection $f : V_C \rightarrow V_M$. Although, in general, a cluster of variables of the sample graph could map to a cluster of variables in the model graph, we assume for this paper that each cluster of sample graph variables maps to only a single model graph variable. More precisely, we assume that each model graph vertix represents a cluster of design variables.

In computing the level of isomorphism between two graphs, the *graph edit distance* [14] is computed between the graphs. The graph edit distance ($\delta : \mathbb{G} \times \mathbb{G} \rightarrow \{0\} \cup \mathbb{N}$, where $\mathbb{G}$ is the set of graphs) is defined as the number of edit operations needed to transform one graph into the other. For this paper, the edit operations are limited to node insertion/deletion and edge insertion/deletion. Given two graphs, G_1 and G_2, $\delta(G_1, G_2) = 0$ only when the graphs are isomorphic.

3. Evolutionary Computing

With the given representation of the problem, we formulate a genetic algorithm [7] whose goal is to find the projection $\pi^* : V_S \rightarrow V_C$ to produce $\vec{C}$ such that $\delta(\vec{C}, \vec{M}) \approx 0$. Meaning that we want to find the clustering of sample graph vertices such that the conformance graph of these clustered nodes is isomorphic, or almost isomorphic, to the model graph.

In a genetic algorithm, potential solutions to the problem are considered as living organisms with genetic material, similar to DNA. In this paper, we refer to these potential solutions as mappings or projections. Each projection π_i is a sequence $\langle \alpha_1, \alpha_2, \ldots, \alpha_{|\mathbb{X}|} \rangle$ of mappings for each vertex of the sample graph to a vertex of the conformance graph. Each α_j is an element from V_C and the projection maps the j-th vertex in V_S to it. The algorithm first creates an initial population $\pi_1, \pi_2, \ldots, \pi_n$ of random projections.

Each projection is judged on how accurate its solution to the problem is, called its fitness. Pairs of projections are selected for "breeding" to produce new projections. These new projections are created by combining the genetic material from their "parent" projections. Only the projections with the highest fitness values survive long enough to breed and pass on their genetic information. The algorithm breeds these projections, one iteration at a time, until a satisfactory solution is found or until a maximum iteration count is reached.

3.1. Fitness

Designers can cluster a DSM in different ways, each representing a perspective of decomposition: directory structure, horizontal layers or vertical slicing [4]. To check the conformance between design and implementation when the design DSM is clustered in a particular way, we need to cluster the source code components accordingly. One feature of a genetic algorithm based approach is that we can configure its *fitness function* to reflect the designer's intentions to cluster a system so that variables within a sample DSM can be clustered accordingly.

In this section we discuss how the fitness function can be configured to control the way the source code components are clustered. Each projection is assigned a fitness based on how accurate its solution is to the problem. For our current approach, the fitness of each organism is calculated using the formula:

$$-\delta(\vec{C}'_i, \vec{M}) - \epsilon(\vec{C}_i, \vec{M}) - \lambda(\vec{C}_i, \vec{M}) - \phi(\vec{C}_i, \vec{M}) \quad (1)$$

where $\vec{C}_i$ is the conformance graph at the i-th iteration of the algorithm.

The fitness function component δ is the graph edit distance, modeling the differences between the dependence structures of the two DSMs. If the graph edit distance is zero, the conformance graph and model graph are isomorphic. In most cases, we can't expect the source code structure to be identical to design structure. Accordingly, our approach does not require the graph edit distance to be zero for the algorithm to terminate. Instead, our algorithm finds the solutions that maximizes the level of isomorphism.

In this paper, we assume that all the design dimensions modeled in a design DSM are implemented in the source code. Based on this assumption, we augment the fitness function with a penalty for projections in which the conformance graph vertices and model graph vertices do not have a one-to-one mapping. We define such a penalty using ϵ, as shown in formula (1).

Using the graph edit distance for the fitness function is not sufficient. Our initial experiments showed that there are multiple ways to cluster the sample DSM that create the same dependency structure. In our algorithm, we added two additional components to the fitness function to provide finer differentiation between mappings with the same graph edit distance. These two functions allow us to configure a sample graph so that it can be clustered in different ways, each corresponding to how the design targeted DSM is clustered.

3.1.1 Directory Groupings

It is often the case that in large software systems, source code components belonging to the same high level design

0-7695-2967-4/07 $20.00

component are grouped into the same file system directory or package. For example, Lattix drives DSMs from Java source code and organizes the derived DSMs so that each block along the diagonal represent a Java package. Accordingly, our approach allows the user to specify the directory groupings of source code components. We use a dissimilarity metric to calculate how separated components from each directory grouping are. Depending on this measure, the fitness of the projection is proportionally reduced. This reduction of the fitness is denoted in formula (1) as λ.

3.1.2 Name Patterns

Using regular expressions to model name patterns to capture a group of components is a well understood approach. Murphy's Relexion [12] uses name patterns to specify mappings between high level models and implementations; in *aspect-oriented programming* (AOP) paradigm, name patterns are used to specify *join points* for the weaving of advice into aspect code. The use of these name patterns show that in many cases, naming patterns exist in source code that represent commonalities between different components [8].

Accordingly, our approach allows the user to specify name patterns as part of the fitness function such that sample graph vertices matching the pattern should be clustered and mapped to a particular model graph vertex. If a sample graph node attribute matches the pattern but is not correctly mapped to the model graph vertex then the fitness of the projection is reduced. This reduction of the fitness is denoted in formula (1) as ϕ.

4. A Canonical System

As a proof-of-concept experiment, we apply our approach to automatically checking the modularity conformance between the design and implementation of Parnas's canonical *Keyword in Context* (KWIC) system [13]. In this paper, we only examine the information hiding design. There are six modules in this design, *LineStorage*, *Input*, *CircularShift*, *Alphabetizer*, *Output*, and *MasterControl*. We first derive DSM models for both the design and the implementation using Simon [5] and Lattix [1], respectively, and then run the algorithm to check their conformance with different fitness configurations.

Being as famous as it is, KWIC has been implemented by many people. We selected a representative version developed at the Institute for Information Processing and Computer Supported New Media (IICM), Graz University of Technology[1], for our analysis.

[1] http://coronet.iicm.edu/sa/swp_how.htm

4.1. Representation

Given the selected implementation, we derived their source code DSM using Lattix, as seen in figure 2, and created the sample graph as seen in figure 3. A high level design DSM was automatically derived from the logical model of Parnas's information hiding design using Simon [5], as shown in figure 1, and the created model graph can be seen in figure 4.

Although in most software systems we expect the model graph to have fewer vertices than the sample graph, the tool we used for deriving the source code DSM, Lattix, only provides class-level abstraction. As the result, the source code DSM has fewer variables than the design level DSM. As a proof of concept for our approach, we created a sample graph using the Simon DSM and a model graph using the Lattix DSM. Our algorithm clusters the sample graph and maps the clustered graph to the model graph. The shaded blocks in the sample graph, figure 4, show the correct projection we expect. For example, the *InputADT* and *InputImpl* vertices from the sample graph map to the *Input* vertex from the model graph.

4.2. Analysis Approach

We created a tool to implement the genetic algorithm, and ran the tool on the two DSM models of KWIC software system. Our hypothesis is that: because the design DSM is validated, and the source code faithfully implemented the design, these two DSM models should be consistent with each other, and our experiment results should find the desired mapping, and confirm their consistency quickly. Our tool ran on a 2.16GHz Intel Core 2 Duo MacBook Pro with 1GB of RAM.

First, we ran our tool on the graphs with five directory groupings specified. Since there were no actual file system directories grouping the separate components, we artificially made up ones as a proof of concept to demonstrate the ability to handle directory groupings. We assumed each variable in the model graph represented a directory, and specified directory groupings as fitness functions to generate correct projections.

Next, we ran our tool on the graphs by specifying five name pattern matchings as fitness function components so that vertices with the same name patterns are grouped as a cluster. There was clearly a naming convention being followed in the sample graph, for example, all variables starting with the string "Input" was mapped to the *Input* model graph vertex. Finally, we ran our tool with both directory groupings specified and name patterns specified.

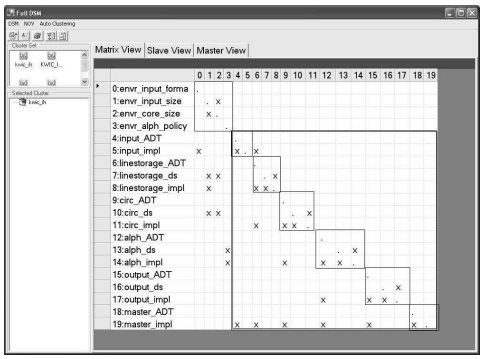

Figure 1. Simon [5] Design DSM

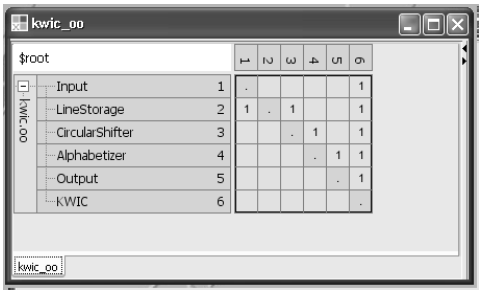

Figure 2. Lattix [1] Source Code DSM

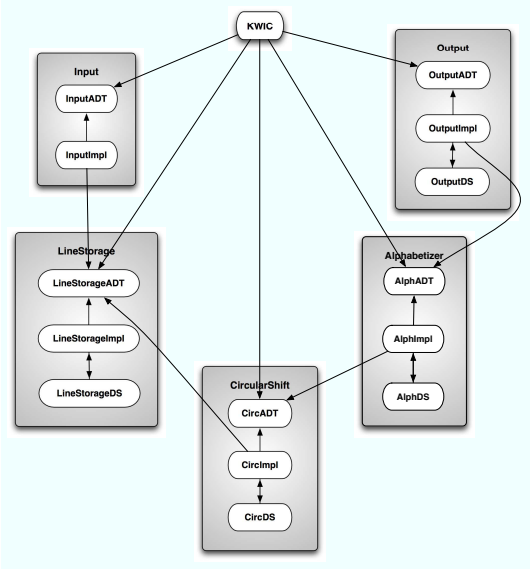

Figure 3. Sample Graph

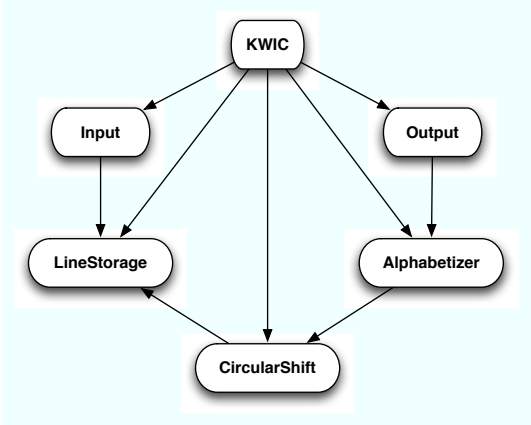

Figure 4. Model Graph

4.3. Analysis Results

We ran our tool on the two KWIC DSMs and checked their consistency in terms of modularity. Our experiments consistently converged to produce the desired result. We configured our genetic algorithm to run 2000 generations, and correct results are constantly found within two minutes on our machine. Our tool sometimes produces a projection that, although would create isomorphic graphs, was not the desired view of the source code. Despite this, most of the runs produced the source code view desired. When both the directory groupings and name patterns were specified the desired projection was more often produced.

Our experimental results confirm to our hypothesis that these two DSM modules are consistent with each other. This experiment also shows the feasibility of using a genetic algorithm to automatically cluster DSM variables and correctly map source code components to high level design components for modularity analysis. Although currently our program only allows the user to specify directory groupings and name patterns and the software system we analyzed is quite small, the fitness function can be extended with additional criteria, and we believe that with a good fitness function for the genetic algorithm, this approach can be scaled to much larger software systems.

5. Related Works

DSMs were created by Steward [16], developed by Eppinger [6], and has been applied to other engineering dis-

ciplines [6, 16]. Baldwin and Clark have sketched a novel theory providing new insights into the connections between design structure and economic value in design using DSM modeling [3]. Sullivan et al. [17, 18], and Lopes et al. [10] have shown that DSM modeling is valuable for software design. For example, a DSM can capture Parnas's information hiding criterion precisely [17].

There are several other approaches and tools for automatic clustering and analyzing software modularity. Typical "bottom-up" tools, such as Bunch [11] and Lattix [5], reverse engineer high level design models from source code. Our automatic clustering approach is different in that we use a high level design model as the optimal goal and our fitness function is configurable. Design level modeling tools, such as Simon [5], support high level logical design modeling, and is not connected with implemented source code.

Our work is similar to Murphy's Reflexion Model [12] in that our approach also combines a source code model with a high level model. Our work is different from Murphy's work in that our approach employs the power of DSM, supports auto-clustering and multi-clustering, and emphasizes modularity checking. Our approach has the potential to link source code structure with design structure and environment conditions, so that we can extend Baldwin and Clark's net option value analysis [3] and Parnas's information hiding analysis [13] to the level of source code.

6. Conclusion

To address the problem that a well modularized design does not always guarantee a well modularized implementation, and the difficulty of checking the conformance between a design modular structure and a source code modular structure manually, we presented an approach to automatically check the consistency between source code structure and design structure. Our approach makes use of design structure matrices as uniform representations, uses existing tools to automatically derive design level and implementation level DSMs, creates a genetic algorithm to cluster DSMs and to check the their consistency by finding the best mapping.

Our approach was quite effective when applied to the small but canonical *keyword in context* software system. The tool automatically clusters the DSMs and confirms the consistency between the two DSMs. Although we have not attempted to apply our approach to large software systems, we believe the approach can be scaled for large software. We plan to continue this work and apply our approach to a large software system such as HyperCast [9].

References

[1] *The Lattix Approach Whitepaper*. Lattix Inc., Nov. 2004.

[2] C. W. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, 1970.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

[4] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.

[5] Y. Cai and K. Sullivan. Simon: A tool for logical design space modeling and analysis. In *20th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2005.

[6] S. D. Eppinger. Model-based approaches to managing concurrent engineering. 2(4):283–290, 1991.

[7] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[9] J. Liebeherr and T. K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication*, pages 72–89, 1999.

[10] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[11] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. *IEEE Proceedings of the 1999 International Conference on Software Maintenance*, Aug. 1999.

[12] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, 1995.

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–8, 1972.

[14] A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):353–362, 1983.

[15] W. P. Stevens, G. J. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–39, 1974.

[16] D. V. Steward. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–84, 1981.

[17] K. Sullivan, Y. Cai, B. Hallen, and W. G. Griswold. The structure and value of modularity in software design. *SIGSOFT Software Engineering Notes*, 26(5):99–108, Sept. 2001.

[18] K. Sullivan, W. Griswold, Y. Song, and Y. C. et al. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE '05*, Sept. 2005.